ELSEVIER

Theoretical Computer Science 154 (1996) 203–224

Theoretical Computer Science

Multiple matching of parameterized patterns☆

Ramana M. Idury[a,1], Alejandro A. Schäffer[b,2,*]

[a] *Department of Mathematics, University of Southern California, 1042 W. 36th Place, DRB 155, Los Angeles, 90089-1113, USA*

[b] *Department of Computer Science, Rice University, 6100 Main Street, Houston, TX 77005-1892, USA*

Received February 1994; revised October 1994

Communicated by M. Crochemore

Abstract

We extend Baker's theory of parameterized string matching (1993) to algorithms that match multiple patterns in a text. We first consider the case where the patterns are fixed and preprocessed once, and then the case where the pattern set can change by insertions and deletions. Baker's algorithms are based on suffix trees, whereas ours are based on pattern matching automata.

1. Introduction

In string matching applications, we are often given a pattern and a text, and asked to find the locations in the text at which the pattern *matches* the text. The match could be exact, in which case we report only those locations where the pattern exactly matches the text. The match could be approximate, in which case we allow a small number of *errors* to occur between the matched portions of the pattern and the text. Typically, errors are defined using the editing distance criterion in many text processing applications.

However, in some text processing applications, a different criterion for defining errors is needed. For example, commands like "global-replace" in the editor emacs allow users to substitute all occurrences of a string x with a string y instantaneously. If

☆ A preliminary version of this paper appeared in the Proceedings of CPM'94, Springer, Berlin, LNCS 807, 1994. Submission of this paper in the final form to Theoretical Computer Science was invited in view of the strong endorsement contained in the conference reviews.

* Corresponding author. Email addresses: idury@hto.usc.edu; schaffer@cs.rice.edu.

[1] Supported by NSF grant DMS-90-05833.

[2] Partially supported by NSF grant CCR-9010534.

SSDI 0304-3975(94)00270-3

there are many occurrences of x in the original text, the new text will contain y at each occurrence and it will not match the old text using the editing distance criterion because there is no fixed limit on the number of substitutions performed.

Global replacements are a common way to reuse code with small modifications. However, code duplicated by copying and global replacements is often associated with bugs and plagiarism, so we would like to be able to match the original code against the duplicated and edited copy using string matching techniques. What is needed is a criterion for matching that allows all occurrences of one string in the pattern to match occurrences of some other string in the text. To address this goal, Baker [7] recently defined a theory of parameterized pattern matching and used her theory to search for duplication in large programs. She uses characters from a special parameter alphabet to denote identical substrings, so for example, x may represent the string *abcbcd* and every occurrence of x represents that same string.

Baker's paper studies the problem of preprocessing a fixed parameterized text T, so as to quickly search fo all occurrences of a single, input parameterized pattern P in the text. The best solution to this problem uses a data structure called a *suffix tree* [20, 17, 8] to preprocess T; Baker uses a variant of this data structure that she calls a *p-suffix tree* to solve the parameterized version of the fixed text searching problem.

In this paper we are concerned with "dual" problems where the patterns are prprocessed and the text is treated as the input (variable). We present algorithms for matching a dictionary D of parameterized patterns against an input text T. In multiple matching, we search simultaneously for different patterns $P_1, P_2, \ldots$ rather than searching the entire text once for each different pattern. We give algorithms both for the case where the dictionary is static and can be preprocessed, and for the case where the dictionary changes over time by the insertion and deletion of individual patterns. We have completely implemented in the language C the algorithms for the more interesting dynamic case.

We start with the following most basic problem.

- *Fixed Pattern Matching* (*FPM*): Given a fixed pattern P of length p over an alphabet Σ, preprocess P so as to be able to find all occurrences of P in a *query* text T of length t.

There are several solutions to FPM that preprocess the pattern in time $O(p)$ and search the text in time $O(t)$. In this paper we are particularly interested in the linear time algorithm of Knuth, Morris, and Pratt (KMP for short) [16] since their automata-theoretic approach can be extended, in a very natural way, to solve the following problem.

- *Multiple pattern matching* (*MPM*): Given a fixed set D of patterns $P_1, P_2, \ldots, P_k$ over an alphabet Σ preprocess D so as to be able to search for all pattern occurrences in a query text T. The set D is sometimes called the *dictionary*.

Aho and Corasick (AC) [1] first solved MPM by generalizing the KMP automaton method to multiple patterns. The AC algorithm preprocesses the pattern dictionary in

time $O(d \log \sigma)$ and searches a text in time $O(t \log \sigma + tocc)$, where d is the total size of all patterns, σ is the number of characters that occur in a pattern, and $tocc$ is the total number of occurrences reported.

Meyer [19] first posed the problem of extending the MPM problem to allow the dictionary to increase over time by inserting single patterns. Amir and Farach [3] first considered allowing the dictionary to change by both insertions and deletions, and obtained the first interesting time bounds. They defined the problem as follows:

- *Dynamic dictionary matching* (*DDM*): Preprocess and maintain a dictionary D of patterns under the operations *insert* a pattern, *delete* a pattern, and *search* for a query text T for all occurrences of patterns currently in the dictionary.

The DDM problem for one-dimensional strings was studied further in [4, 15, 5] and we summarize the best bounds known for the case of an alphabet of arbitrary size.

Preprocessing: $O(d \log \sigma)$;

Insertion/Deletion: $O(p \log d)$, where p is the length of the pattern;

Text Scanning: $O((t + tocc)\log d)$.

Other search/update time combinations are achievable [15]. Slightly better time bounds are achievable when the alphabet is finite [5]. One of the themes of this paper is the contrast between suffix tree algorithms and automata algorithms. Therefore, it is interesting to note that the DDM algorithms in [3, 4] use suffix trees, the algorithm in [15] uses automata methods, and the best asymptotic algorithm in [5] combines features of the two approaches.

We now summarize Baker's definitions for the parameterized pattern matching problem. As in the above problems Σ will be the base alphabet of characters. There is a second alphabet Π, called the *parameter alphabet* which is disjoint from Σ. The two alphabets Σ and Π are assumed not to contain any positive integers. A string of symbols over $\Sigma \cup \Pi$ is called a *parameterized string* or *p-string*. In our examples, we shall use $\Sigma = \{a, b, c\}$, $\Pi = \{x, y, z\}$; a sample p-string is *axybcxxzbcx*.

Two p-strings x and y *p-match* if x can be transformed into y by applying a one-to-one function on $\Sigma \cup \Pi$ that is the identity on Σ. For example *yyzbcy* p-matches *axybcxxzbcx* at the right end. We can now combine the notion of parameterized matches with the three problems above.

- *Parameterized fixed pattern matching* (*PFPM*): Given a fixed p-pattern P of length p over the alphabet $\Sigma \cup \Pi$, preprocess P so as to be able to find all p-matches of P in a *query* text T of length t.

- *Parameterized multiple pattern matching* (*PMPM*): Given a fixed set D of p-patterns $P_1, P_2, \ldots, P_k$ over the alphabet $\Sigma \cup \Pi$, preprocess D so as to be able to search for all p-matches in a query text T.

- *Parameterized dynamic dictionary matching* (*PDDM*): Preprocess and maintain a dictionary D of p-patterns under the operations *insert* a p-pattern, *delete* a p-pattern,

and *search* for a query text T for all occurrences of the patterns currently in the dictionary.

In Section 2, we review the AC automaton and prove that it can be modified to also solve the PMPM problem within the same time bounds as the AC algorithm. An extension of the KMP automaton to solve the PFPM problem will follow as a special case. Baker (personal communication) also noted that it is possible to extend the KMP automaton to solve the PFPM problem. Amir et al. [6] have recently provided some lower bounds for parameterized pattern matching.

A more interesting question is whether it is possible to extend the automata approach to solve the PDDM problem which combines the dynamic dictionary paradigm with the notion of parameterized matches. From a theoretical perspective, it is important to try as much as possible to combine different extensions to the basic FPM problem, so as to derive string matching algorithms that are as general as possible. From a practical perspective, the dynamic dictionary paradigm applies naturally to the code duplication problem that motivated the definition of parameterized matching. The dictionary of patterns could be small pieces of Program 1 that we would want to match to Program 2. If we change those parts of Program 1 that we want to match, or have multiple versions of Program 1, then we get an instance of the PDDM problem.

Our main result, given in Section 3, is that it is possible to design an automaton algorithm to solve the PDDM problem. Our time bounds for the deletion and search are the same as for the DDM problem. Inserting a pattern takes time $O(p(\log^2 d + \log \sigma))$. One interesting feature of our PDDM algorithm is that we work with two dual representations of a parameterized pattern and one of the representations is computed only *implicitly* but never stored or printed.

Our implementation of the PDDM algorithm includes as a separate package an implementation by the first author [14] of the O(1) amortized-time solution to the *order maintenance problem* due to Dietz and Sleator [10]. Their data structure maintains a list under the operations: insert an item in a given gap, delete an item, and compare the relative order of two items. They also gave a solution that does each of the operations in *worst-case* O(1) time, but the version that achieves the amortized O(1) bounds appears much more practical. The Dietz–Sleator data structure arises in the solution of other dynamic problems [2, 12, 11].

2. Parameterized multiple pattern matching

In this section, we review the essence of the AC algorithm and show how to extend it to solve the parameterized multiple pattern matching problem (PMPM). The AC algorithm scans the text one character at a time computing for each text prefix, the longest prefix of a pattern that matches a suffix of the text scanned so far. If the

matching pattern prefix has a full pattern as a suffix, then all full patterns that are suffixes are output.

The AC algorithm uses an automaton whose states are precisely the pattern prefixes. We use a state and the corresponding pattern prefix interchangeably; we call the pattern prefix the *label* of its state. There are two partial functions, *goto* and *fail*, that represent the forward and backward automaton transitions. There is a third function *output*, such that for any state x, *output*(x) is the list of all patterns that are suffixes of x.

For state x and character a, define *goto*$(x, a) = xa$, if state xa exists and let it be undefined if state xa does not exist. For the empty start state, ε, define *fail*$(\varepsilon) = \varepsilon$. For a nonempty state x, define *fail*(x) to be the longest proper suffix of x that is a state in the AC automaton. The *goto* transitions form an outward tree whose root is ε; the *fail* transitions form an inward tree whose root is ε [1].

The AC search algorithm is described by the following basic loop and output algorithm:

```
SEARCH(T = t_1...t_n)
  state ← ε
  for i ← 1 to n do
    While goto(state, t_i) is undefined
      state ← fail(state)
    state ← goto(state, t_i)
    If output(state) is nonempty
      print location i and output(state)
```

For any given choice of *state* and *symbol* we may have to take the *fail* transition repeatedly, but this shortens the length of the new *state*. The total time needed to scan a text T using this algorithm is $O(t(g+f))$, where $g+f$ is the time needed to make one evaluation of *goto* and *fail* [1].

Baker defined a procedure *prev* to convert a string S on $(\Sigma \cup \Pi)$ to a string of the same length on $(\Sigma \cup \mathbb{N})$ that helps simplify finding p-matches. The procedure *prev* is defined from left to right on the characters of S. For each $c \in \Sigma$, *prev* maps the character to itself. For each $c \in \Pi$, *prev* maps the first occurrence of c to 0 and each successive occurrence of c to the number of characters since the previous occurrence. For example, *prev*$(axbybbyxyxay) = a0b0bb3622a3$. Baker defined a *p-suffix* of S to be a string obtained by applying *prev* to a suffix of S. Also define *psuffix*$(S, i) =$ *prev*$(S[i \ldots n])$, where $n = |S|$. The utility of *prev* is that:

Lemma 2.1 (Baker [7]). *P-strings* S_1 and S_2 *p-match if and only if* *prev*$(S_1) =$ *prev*(S_2).

Lemma 2.2 (Baker [7]). *If* P *is a p-string pattern and* T *is a p-string text, then* P *p-matches* T *starting at* $T[i]$ *if and only if* *prev*(P) *is a prefix of* *psuffix*(T, i).

Example 2.3. Suppose $P = axbxy$ and $T = byaybyxcc$. Then $prev(P) = a0b20$ and $prev(T) = b0a2b20cc$. *Also*, $psuffix(T, 3) = a0b20cc$ and P p-matches T at location 3.

We shall assume that the patterns in D and text are given as strings on $\Sigma \cup \mathbb{N}$ that represent *prev* applied to the original patterns and text. Thus in Example 2.3, we would assume that P is presented as $a0b20$ and T is presented as $b0a2b20cc$. As explained in [7] *prev* can be computed from left to right in linear time, so this is a reasonable assumption. This assumption is used only to simplify the theoretical exposition; in our implementation the strings are on the alphabet $\Sigma \cup \Pi$ and we choose disjoint subsets of ASCII to be Σ and to be Π.

The set of states in our automaton will be the set of prefixes of strings in D. For a string $u = u_1 \ldots u_m$ on $\Sigma \cup \mathbb{N}$, define $shorten(u)$ character by character as follows. Each character in Σ is mapped to itself by *shorten*. If $u_i = r \in \mathbb{N}$ then *shorten* maps u_i to r if $r < i$, and maps u_i to 0 if $r \geqslant i$. For example, $shorten(a3b27b3c90) = a0b20b3c00$.

Lemma 2.4. *Let* $W = w_1, \ldots, w_m$ *be a string on* $\Sigma \cup \Pi$ *and let* $U = u_1, \ldots, u_m = prev(W)$. *Let* W_i *be the suffix* $w_i, \ldots, w_m$ *and let* U_i *be the suffix* $u_i, \ldots, u_m$. *Then* $prev(W_i) = shorten(U_i)$

Proof. The proof is by induction on the length of W_i. Let c be an arbitrary character (occurrence) in W_i. Assume as inductive hypothesis, that $prev(W_i)$ agrees with $shorten(U_i)$ on all the characters before this c.

If $c \in \Sigma$, then both *prev* and *shorten* will map c to itself. If $c \in \Pi$ and c occurs as the jth character in W_i, suppose that when *prev* was applied to W, *prev* mapped this occurrence of c to the value r. If $r = 0$, then this is the first occurrence of c in both W and W_i, and *shorten* maps the 0 to itself. If $r > 0$, then there is a preceding occurrence of c in W and *prev* maps this occurrence to the distance between it and its predecessor, which is r. If $r < j$, then the predecessor is in W_i; therefore, the value of the function *prev* when applied to W_i for this occurrence of the character c is r, which is the same value that *shorten* assigns to the character in the corresponding position in U_i. If $r \geqslant j$, then the preceding occurrence of c in W is not in W_i; therefore, this is the first occurrence of c in W_i and the value of the function *prev* for this occurrence of the character c in W_i is 0, and the function *shorten* maps the value r in U_i to 0. The lemma follows by induction. □

Corollary 2.5. *Pattern* P *matches substring* S *of the text if and only if* $prev(P) = shorten(S)$.

We extend the definitions of *goto, fail, output* to *pgoto, pfail, poutput*. For $a \in \Sigma$ and $x = \varepsilon$ define $pgoto(x, a)$ to be a if that state exists and ε otherwise. For $a \in \mathbb{N}$ and $x = \varepsilon$ define $pgoto(x, a) = 0$ if state 0 exists, and ε if state 0 does not exist. For $a \in \Sigma$ and $x \neq \varepsilon$, define $pgoto(x, a) = xa$, if state xa exists; otherwise $pgoto(x, a)$ is undefined. If $a \in \mathbb{N}$ and $x \neq \varepsilon$ let $q(x, a) = 0$ if $a > |x|$ and let $q(x, a) = a$ is otherwise. Then

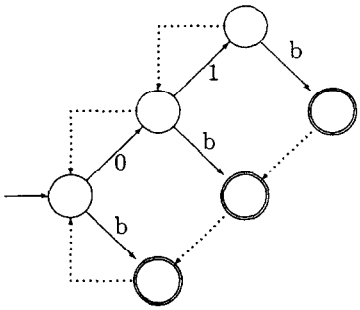

Fig. 1. A sample automaton for the p-patterns $\{b, 0b, 01b\}$; *pgoto* transitions are drawn solid and *pfail* transitions are drawn dotted.

$pgoto(x, a)$ is $x \cdot q(x, a)$ if that state exists and undefined otherwise. Define $pfail(x)$ to be the longest proper suffix y such that $shorten(y)$ is a state in the automaton. Fig. 1 shows a sample automaton.

From the definitions above and the previous lemma, we get the following characterization of *pfail*.

Corollary 2.6 *Suppose that U is an automaton state, $U = prev(W)$ and W_i is the longest proper suffix of W such that $prev(W_i)$ is a state in the automaton. Then $prev(W_i) = shorten(U_i) = pfail(U)$.*

For each state x, $poutput(x)$ is the set of patterns P such that P equals *shorten* applied to some suffix of x.

The search algorithm to solve the PMPM problem is very similar to the version of the AC algorithm given above; we need a new correctness proof because *pgoto* and *pfail* are fundamentally different from *goto* and *fail*.

```
SEARCH(T = t_1 ... t_n)
  state ← ε
  for i ← 1 to n do
    While pgoto(state, t_i) is defined
      state ← pfail(state)
    state ← pgoto(state, t_i)
  If poutput(state) is nonempty
      print location i and poutput(state)
```

Lemma 2.7. *The search loop maintains the invariant that after t_i is scanned, state is the longest state such that there is a suffix x of $t_1 \ldots t_i$ with state = $shorten(x)$.*

Proof. The proof is by induction on the number of characters scanned. The initial state is $\varepsilon = shorten(\varepsilon)$. Suppose the claim holds for the first $i - 1$ characters of T, and we scan t_i.

Let $state_0$ be the state we are in just before we scan t_i. Let *tmatch* be the suffix of length $state_0$ of $t_1, \ldots, t_{i-1}$. By induction hypothesis, $state_0 = shorten(tmatch)$.

Let $state_1 = pfail(state_0)$, $state_2 = pfail(state_1)$, ... be the sequence of states assigned to the variable *state* while considering t_i. By Corollary 2.6, we can define a corresponding sequence of proper suffixes of $tmatch = tmatch_0$, say $tmatch_1, tmatch_2, \ldots$, such that for each $j \geqslant 1$, $shorten(tmatch_j) = state_j$, and $tmatch_j$ is the *longest* suffix of $tmatch_{j-1}$ such that *shorten* applied to this suffix is an automaton state.

In the while loop, we seek the first $state_j$ in the sequence that can be extended via *pgoto* to match the next character t_i. If $t_i \in \Sigma$, then we want to move to state $state_j \cdot t_i$ and that corresponds to the definition of *pgoto*. If $t_i \in \mathbb{N}$, what we do depends on whether $t_i > |tmatch_j|$.

Suppose $t_i > |tmatch_j|$. Then we want to go to $state_j \cdot 0$ if it exists. This is the state that *pgoto* looks for. Suppose $t_i \leqslant |tmatch_j|$. Then we want to go to state $state_j \cdot t_i$ if it exists, and this is the state that *pgoto* looks for. Since we try the suffixes $tmatch_j$ in decreasing order of length, the new state will be obtained from the longest suffix $tmatch_j$ that can be extended with t_i to match the prefix of some pattern. The invariant in the lemma follows by induction on i. □

Lemma 2.8. *The time to scan the text is* $O(t \log \sigma + tocc)$, *where* σ *is now the number of regular and parameterized characters occurring in the original patterns.*

Proof. The analysis of running time is similar to that used for the AC algorithm. Each successful application of *pgoto* extends the length of the matched string by at most one. There are n successful applications of *pgoto*, one per text character. Each application of *pfail* decreases the length of the matched string by at least one. Hence the number of applications of *pfail* is at most n.

The symbols associated with the outgoing *pgoto* transitions are in $\Sigma \cup \mathbb{N}$, so the number of possible outgoing transitions from any state is at most σ because each parameter character can account for at most one distinct outgoing edge. By storing the transition symbols in a balanced tree, the time to evaluate *pgoto* is $O(\log \sigma)$, and the overall time bound $O(t \log \sigma + tocc)$ follows. □

Corollary 2.9. *If there is only one pattern (as in the PFPM problem), the search time can be improved to* $O(t)$.

Proof. When there is only one pattern, there is only one *pgoto* transition out of each nonempty state, so we do not need the binary search. When there is only one pattern $tocc \leqslant t$. □

We have completed the description of the search algorithm using the automaton. What remains is to construct the automaton states, and especially the transition functions, *pgoto* and *pfail*. The following algorithm constructs the automaton states and *pgoto* by adding one pattern at a time. Recall that each pattern is presented in *prev* notation.

```
BUILD-PGOTO(D = {P_1, ..., P_k})
  Create the empty state ε
  for i ← 1 to k do
  /*Insert pattern P_i = P_i[1 ... l_i]*/
    state ← ε
    j ← 1
    newstate ← false
    do/* go through existing states*/
      if pgoto(state, P_i[j]) is defined then
        state ← pgoto(state, P_i[j])
        j ← j + 1
      else
        newstate ← true
    while newstate = false and j ⩽ l_i
    for r ← j until l_i do
      create state state · P_i[r] storing its label length
      pgoto(state, P_i[r]) ← state · P_i[r]
      state ← state · P_i[r]
    add P_i to poutput(state).
  for all c ∈ Σ ∪ {0} such that pgoto(ε, c) is undefined
    pgoto(ε, c) ← ε
```

As in [1], we do not actually store the state labels but instead, number the states consecutively. Unlike [1] we do store the length of the label, so that we can compute *shorten* quickly.

Lemma 2.10. *The algorithm to construct pgoto is correct and runs in* $O(d \log \sigma)$ *time.*

Proof. The correctness proof is a straightforward induction on the number of patterns. Suppose that the *pgoto* function is correct for $D = \{P_1, \ldots, P_{i-1}\}$ and we are about to insert P_i. The do-while loop finds the longest prefix U_i of P_i such that $shorten(U_i)$ is a state that already exists. In the for loop on r we scan the remaining characters of P_i adding one state per character. For each character we extend the state label by appending that character and update *pgoto* accordingly. In the construction of *pgoto*, alphabetic characters and numeric characters can be treated in the same way because we assume that the pattern is in *prev* notation.

The total number of iterations of the do-while loop and the for loop on r is d. The length of the new state $state \cdot P_i[r]$ is $1 + |state|$ and can be computed in constant time. This means that creating a new state takes $O(1)$ time. Updating *pgoto* corresponds to an insertion in a balanced tree, which takes time $O(\log \sigma)$. We can store *pgoto* for state ε implicitly by storing a tree for only those values that are not ε. The total time is $O(d \log \sigma)$. □

Each state has one incoming *pgoto* edge whose symbol is the last character of the state name, so the *pgoto* edges form an outgoing tree rooted at ε.

To compute *pfail* and fill in *poutput* we visit the states in the *pgoto* tree in breadth-first order. To get started we set $pfail(c) \leftarrow \varepsilon$ for all states c with one character names. If we have computed $pfail(z)$ for all states z such that $|z| < |x|$ we can compute $pfail(x)$ using:

```
BUILD-PFAIL(x)
  Suppose x = yc /* c is the last character of x*/
  found ← false
  do
    temp ← pfail(y)
    if pgoto(temp, c) is defined
      pfail(x) ← pgoto(temp, c)
      found ← true
    else if temp = ε
      pfail(x) ← ε
      found ← true
    y ← temp
  while found = false
  poutput(x) ← poutput(x) ∪ poutput(pfail(x))
```

Lemma 2.11. *If* build-pfail(x) *is applied in increasing order of length of* x, *then all the values of pfail are computed correctly. The time to compute all values is* $O(d \log \sigma)$.

Proof. Let x be an arbitrary state whose label is of length at least 2. Assume as inductive hypothesis, that *pfail* has been computed correctly for all shorter states. Let $x = yc$. By Corollary 2.6, we need to find the longest proper suffix u of x such that $shorten(u)$ is an automaton state. For any string $w = w'c$, $shorten(w) = shorten(w')\,0$ if $c \in \mathbb{N}$ and $c > |w'|$ and $shorten(w) = shorten(w')c$, otherwise. Thus we need to find the longest suffix y' of y such that:

(1) $shorten(y')$ is an automaton state,

(2) $pgoto(shorten(y'), c)$ is defined.

If no suffix satisfying both conditions exists, then $pfail(x) = \varepsilon$.

Since we assumed that the smaller values of *pfail* are correct, it follows from Corollary 2.6 that the sequence of suffixes that satisfy condition (1) is given by $pfail(y)$, $pfail(pfail(y)), \ldots$ The do-while loop finds the first such suffix that satisfies condition (2). If no satisfactory suffix is found, the algorithm correctly sets $pfail(x) \leftarrow \varepsilon$.

The running time analysis is similar to that for searching. For each pattern P, we account for the cost of computing *pfail* for all prefixes of P together. Since each computation of build-pfail gives a shorter string, we can evaluate *pfail* and *pgoto* at most $|P|$ times in all do-while loops among all the calls for prefixes of P. Each evaluation of build-pfail takes $O(|P|)$ time. Each evaluation of *pgoto* takes

$O(\log \sigma)$ time. Summing over all patterns, gives the desired time bound $O(d \log \sigma)$. □

Lemma 2.12. *The initialization of poutput at the end of* build-pgoto *and the increment at the end of* build-pfail *ensure that for each state x, poutput(x) contains precisely those patterns P such that P equals shorten applied to a suffix of x.*

Proof. For each state x, we put x in *poutput*(x) at the end of the build-goto if and only if x is a full pattern. The elements of *poutput*(x) that are shorter that x are inserted in build-pfail. We prove that this is correct by a simple induction.

Let x be some arbitrary prefix, and assume that *poutput*(y) has been computed correctly for all y such that $|y| < |x|$. Let $z = pfail(x)$. By Corollary 2.6 and Lemma 2.11 z is the *longest* automaton state that equals *shorten* applied to a suffix of x. For any suffix x' of x such that $|x'| \leqslant |z|$ let the suffix of z of length $|x'|$ be z'. From the definition of shorten it follows that $shorten(x') = shorten(z')$. Thus any pattern of length $\leqslant |z|$ equals *shorten* applied to a suffix of z if and only if it equals *shorten* applied to the same length suffix of x. Therefore, the set of patterns shorter than x that belong to *poutput*(x) are precisely those in *poutput*(z). □

In sum:

Theorem 2.13. *It is possible to solve the PMPM problem using preprocessing time* $O(d \log \sigma)$ *and scanning time* $O(t \log \sigma + tocc)$

3. Dynamic dictionary matching of parameterized patterns

To modify the previous algorithms for the dynamic case we must be able to compute *pgoto*, *poutput*, and *pfail* when the set of patterns is changing. Managing *pgoto* is straightforward and will be discussed briefly. In the dynamic setting we will not be able to store the *poutput* function; instead we reuse a technique for the DDM algorithm in [15] to do the output on the fly. To compute the *pfail* function we would like to use the solution in [15, 5], but it does not work for numerical characters.

As in the AC algorithm and the algorithm in the previous section, we store the *pgoto* function as a directed tree rooted at ε, where the nodes correspond to automaton states, whose implicit labels are the value of *prev* applied to a pattern prefix. Every edge is labeled with a character from $\Sigma \cup \mathbb{N}$, although we shall later augment Σ to compute the output and compute *pfail*. The fact that the dictionary is dynamic has minimal impact on the way the *pgoto* function is computed, and we can still use the algorithm from the previous section to update *pgoto* when patterns are inserted. We store with each state a pointer to its parent state in the *pgoto* tree. For each state x we keep a count of how many patterns P have x as a prefix. When a pattern is deleted, we visit the states for prefixes of the pattern from longest to shortest. For each prefix of

the pattern, we decrement the pattern counter of that state and delete the state if the counter decreases to 0 (meaning that no patterns other than the deleted one use that state).

To recognize patterns and compute the output, we make two modifications to the pattern inputs. First, we assume that the empty string ε is always a pattern in the dictionary, although we never output ε as a matched pattern. Second, we add a distinct symbol \$ to Σ and we assume that \$ is the largest symbol in the lexicographic order on Σ. We append to each pattern the special symbol \$ before inserting the pattern. We use the symbol $D\$$ to represent the dictionary with \$ appended to every pattern. For each pattern P, there will be a state corresponding to the string $P\$$ at which we recognize P; we keep a pointer to the pattern at that state to be able to print the pattern. We extend the definition of *pgoto*, so that $pgoto(P, \$) = P\$$, for each pattern P. We use the following dictionary as an example to explain various definitions and concepts of our automaton.

Example 3.1. Suppose $\Sigma = \{a, b, \$\}$. Let $\hat{D}\$ = \{\$, b\$, b0\$, a0b2\$\}$ be a sample dictionary where every pattern is appended with the special symbol \$.

We call a state label a *normal prefix* if it is a prefix of some pattern in $D\$$. For each proper prefix x of a pattern we also define an *extended prefix*, $x\$$, by appending the character \$. The corresponding states are called normal or extended. In Example 3.1, the set of normal prefixes is $\{\varepsilon, a, a0, a0b, a0b2, b, b0, \$, b\$, b0\$, a0b2\$\}$ and the set of extended prefixes is $\{\$, b\$, b0\$, a\$, a0\$, a0b\$, a0b2\$\}$. Notice that some prefixes are both normal and extended. In this example, $pgoto(b0, \$) = b0\$$ because $b0$ is a pattern, but $pgoto(a0b, \$)$ is not defined because $a0b$ is not a full pattern.

We modify the definition of *pfail* to accommodate the extended prefixes as follows: Let w be a state label (normal or extended). Define $pfail(w) = shorten(x)$ such that $|x| < |w|$ and x is the longest suffix of w such that $shorten(x)$ is a *normal* prefix. In Example 3.1, $pfail(a0b2\$) = b0\$$ because $b0 = shorten(b2)$.

We recognize patterns as follows. When we reach a position of a text, and compute *pgoto* for that position, we pretend that the next symbol is a \$. If we can make another *pgoto* transition to some *normal* prefix ending with a \$, then we know that a pattern has been matched at that position, since any normal prefix ending with a \$ must be a pattern in the dictionary. By applying *pfail* repeatedly, we can report all the matching patterns in the order from the longest pattern to the shortest. When we reach the pattern $\varepsilon\$$, we stop without reporting that empty pattern.

Suppose we are searching the text $b0a2b2aa$ for the occurrences of the patterns in the dictionary $\hat{D}\$$ of Example 3.1. After reading the prefix $b0a2b2$ we will be in the normal state $a0b2$. When we pretend to read \$ as the next symbol, we will *temporarily* enter a state $a0b2\$$ and since this is a normal state in the dictionary we report that pattern $a0b2$ is recognized. If we take $pfail(a0b2\$) = b0\$$ we see that we have matched another smaller pattern $b0$. Again, if we take $pfail(b0\$) = \$$ we realize that no more patterns can be matched as this corresponds to the empty pattern ε. By remembering

the state $a0b2$, where we started looking for patterns, we can continue our search by reading the next symbol a from the next.

The basic difficulty in extending the automata approach to the (non-parameterized) DDM problem is how to maintain the *fail* function when doing insertions and deletions. We summarize the key ideas used to maintain *fail* for the DDM problem, so that we can significantly modify them to handle *pfail* in the parameterized version.

Unfortunately, the alphanumeric *prev* prefixes we used as the state labels in the previous section do not have the property that $pfail(x)$ is always a suffix of x. To solve the suffix problem we define a new labeling function called $next:(\Sigma \cup \Pi)^* \rightarrow (\Sigma \cup \mathbb{N})^*$ which can be viewed as a "dual" of *prev*. The function *next* maps each character in Σ to itself. The function *next* maps all but the last occurrence of $c \in \Pi$ to the number of characters until the next occurrence of c. The last occurrence of $c \in \Pi$ is mapped to 0 by *next*. For example, $next(axbxaybbyxy) = a2b6a3bb200$.

The function *next* can be used to define a different labeling scheme for the states in the automaton. The pseudocode below gives an algorithm to convert $prev(x)$ to $next(x)$. One useful property of *prev* is that *prev* is defined from left to right, so that $prev(x)$ is a prefix of $prev(xa)$. Analogously, *next* is defined from right to left, so $next(x)$ is always a suffix of $next(ax)$. For these reasons, the *prev* label is useful for working with *pgoto*, but the *next* label is useful for working with *pfail*.

```
PREV-TO-NEXT(p_1, ..., p_r)
  for i ← 1 until r do
    n[i] ← 0;
  for i ← 1 until r do
    if p_i ∈ Σ
      n[i] ← p_i
    if p_i ∈ ℕ and p_i > 0
      n[i − p_i] ← p_i
  return n
```

Lemma 3.2. *Let w be a string in $(\Sigma \cup \Pi)^*$ and let $p = prev(w)$ and $n = next(w)$. Then* $n =$ PREV-TO-NEXT(p).

Proof. For each character $c \in \Sigma$ all three functions *prev*, *next*, and PREV-TO-NEXT, map c to itself; so all the occurrences of c in w will be preserved in n, p and PREV-TO-NEXT(p). For each character $c \in \Pi$, the function *prev* maps the occurrences (from left to right) of c in w to a sequence of integers $0, w_2, w_3, w_4, \ldots$, where, for $i > 1$, the ith occurrence of c is w_i characters after the preceding occurrence. The function *next* converts the same sequence of occurrences to the characters $w_2, w_3, w_4, \ldots, 0$. Thus in comparison to *prev*, *next* shifts each w_i to the left by w_i characters, except for the last w_i, which is instead a 0. The first loop above assigns all occurrences of c to the default integer 0. The second loop takes all occurrences that *prev* maps to w_i and places that value w_i places

further to the left as *next* would do. Since c is arbitrary, PREV-TO-NEXT handles all characters in $\Sigma \cup \Pi$ correctly. □

Since $next(x)$ is a suffix of $next(ax)$ we can now restate the characterization of *pfail* in Corollary 2.6.

Corollary 3.3. *For any automaton state x, $pfail(x)$ is the longest automaton state y such that* PREV-TO-NEXT(y) *is a proper suffix of* PREV-TO-NEXT(x).

The preceding corollary suggests that we should use the *next* labeling of states to compute *pfail* as follows. We define a total ordering on the *next* labeling of states and their complements and call it the *inverted order* denoted by $<_{inv}$.

Let $\# \notin \Sigma$ be a new symbol such that $\# > a$ for any $a \in \Sigma$ in the lexicographic order; in particular, $\# > \$$. For every prefix $w \in \Sigma^*$ we define $\#w$ as the *complement* of w. We call w a *regular prefix*, and $\#w$ a *complementary prefix*. Let S be the set of regular and complementary prefixes of patterns. We define an ordering $<_{inv}$ on S, such that the regular prefix x is a suffix of the regular prefix y if and only if $x <_{inv} y <_{inv} \#x$. An ordering that satisfies this property has the virtues that

Lemma 3.4 (Idury and Schäffer [15]). *Let $w, x \in S$ be arbitrary regular prefixes.*

1. $w <_{inv} x <_{inv} \# w$ *if and only if* $w <_{inv} \#x <_{inv} \#w$.
2. *If we replace a regular prefix with a "(" and its complement with a ")" then the prefixes of S in the $<_{inv}$ order yield a list of well-balanced parentheses.*

If we store the parenthesis list of part 2 of Lemma 3.4, then $fail(y)$ can be computed by finding the left parenthesis $(_y$ that y is mapped to, finding nearest enclosing parentheses of $(_y$, and then finding the string x that has been mapped to the nearest enclosing left parentheses [15, 5].

For technical reasons needed later, we assume that any $c \in \Sigma$ is $>_{inv}$ than any $m \in \mathbb{N}$ in the ordering of characters. We assume that the numbers are ordered in the opposite of their usual order, so $0 >_{inv} 1 >_{inv} 2, \ldots$ For two distinct strings w and x, $w <_{inv} x$ if $(\text{PREV-TO-NEXT}(w))^R$ comes before $(\text{PREV-TO-NEXT}(x))^R$ in the lexicographic ordering on $(\Sigma \cup \mathbb{N})^*$, where x^R is the reverse of the string x.

Since PREV-TO-NEXT(x) is a suffix of PREV-TO-NEXT(ax), it follows that a regular prefix x is a suffix of a regular prefix y if and only if $x <_{inv} y <_{inv} \#x$. Lemma 3.4 applies to our $<_{inv}$ ordering.

Example 3.5. Consider the list of normal and extended prefixes of $\hat{D}\$$ in Example 3.1. Adding in the complementary prefixes, the elements, in *prev* representation, of the resulting set S are ordered as follows according to $<_{inv}$: ε, $a0$, $\#a0$, $b0$, $a0b2$, $\#a0b2$, $\#b0$, a, $\#a$, b, $a0b$, $\#a0b$, $\#b$, \$, $a0\$$, $\#a0\$$, $b0\$$, $a0b2\$$, $\#a0b2\$$, $\#b0\$$, $a\$$, $\#a\$$, $b\$$, $a0b\$$, $\#a0b\$$, $\#b\$$, $\#\$$, $\#$.

If we convert each prefix to its *next* label, the order of corresponding prefixes is ε, $a0$, $\#a0$, $b0$, $a2b0$, $\#a2b0$, $\#b0$, a, $\#a$, b, $a0b$, $\#a0b$, $\#b$, \$, $a0\$$, $\#a0\$$, $b0\$$, $a2b0\$$, $\#a2b0\$$, $\#b0\$$, $a\$$, $\#a\$$, $b\$$, $a0b\$$, $\#a0b\$$, $\#b\$$, $\#\$$, $\#$ and we can check the order more easily.

The number of extended prefixes is at most equal to the number of normal prefixes, so the number of regular prefixes is $O(d)$. The number of complementary prefixes is exactly equal to the number of regular prefixes. Therefore, the total number of prefixes in our dictionary structure is only $O(d)$, independent of Σ and Π.

To compute *pfail* on the regular prefixes we use the parenthesis mapping suggested in part 2 of Lemma 3.4. More specifically, each regular prefix is mapped to a left parenthesis. The complement of each regular prefix is mapped to a right parenthesis. Each left parenthesis that corresponds to prefix x has a bidirectional pointer between it and the state s_x in the automaton that represents prefix x. To compute $pfail(y)$ for a regular prefix y we do the following steps:

1. Let $(_y$ be the left parenthesis that the state for y points to.
2. Find the nearest enclosing parenthesis pair $(_z,)_{\#z}$ of $(_y$.
3. Find the state z that $(_z$ points to.
4. $pfail(y)$ is z.

The improved algorithm for parenthesis queries in [5] is not applicable because in parameterized matching the alphabet of the prev labels includes $\mathbb{N}$ and is therefore unbounded.

We use a height balanced tree, such as an a–b tree [18] with a distinct leaf representing each prefix in S, including the extended and complementary prefixes to store the list. We assume that the leaves are linked, so that each leaf can find its neighbors in constant time. The a–b tree enables us to insert or delete a new prefix in $O(\log d)$ comparisons and $O(\log d)$ other operations. In our implementation we use a 2–3 tree.

We supplement each node v of the tree with the values of two functions BALANCE(v) and SUFF(v) to be defined shortly. The function BALANCE was first proposed by Güting and Wood in their data structure called a *parenthesis tree* [13].

The parenthesis tree maintains a list of d well-balanced parentheses as its leaves and the parenthesis tree supports the operations:

P_Insert(p_1, p_2, p'_1, p'_2): Insert the matching pair (p_1, p_2) with p_1 immediately after p'_1 and p_2 immediately before p'_2.

P_Delete(p_1, p_2): Delete the matching pair (p_1, p_2).

P_Nearest(p_1): Return the nearest enclosing parentheses of p_1.

Each operation takes $O(\log d)$ worst-case time [13].

In the parenthesis tree of Gütting and Wood the parentheses are represented as leaves in a balanced binary tree. In addition, each pair of matching parentheses are connected to each other by a separate link. The parenthesis tree is augmented with a BALANCE information in the following way: for every node p in the tree, let the pair

BALANCE$(p) = \langle close(p), open(p)\rangle$ denote the number of unmatched closing and unmatched opening parenthesis in its subtree.

For each node p of the tree, the BALANCE(p) can be computed recursively as follows:

(a) *p is a leaf.* Then

$$\text{BALANCE}(p) = \begin{cases} \langle 0,1\rangle & \text{if } p \text{ is ``(''} \\ \langle 1,0\rangle & \text{if } p \text{ is ``)''.} \end{cases}$$

(b) *v is an internal node with children v_1 and v_2.* We compute

$$\text{BALANCE}(v) = \text{BALANCE}(v_1) \oplus \text{BALANCE}(v_2),$$

where

$$\langle i,j\rangle \oplus \langle k,l\rangle = \begin{cases} \langle i, j-k+l\rangle & \text{if } j \geqslant k, \\ \langle i-j+k, l\rangle & \text{otherwise.} \end{cases}$$

We can extend the scheme of Güting and Wood to work with trees with multiple arity (>2), by observing that the $\oplus$ operator is associative. Thus for a node v with children $v_1, v_2, \ldots, v_k$, BALANCE (v) = BALANCE$(v_1) \oplus$ BALANCE$(v_2) \oplus \cdots \oplus$ BALANCE(v_k).

Now we define SUFF. Let x_1, x_2 be two state labels in *prev* notation; then SUFF$(x_1, x_2) = 1 +$ length of the longest common suffix of PREV-TO-NEXT(x_1) and PREV-TO-NEXT(x_2). For example, if PREV-TO-NEXT(x_1) is a suffix of PREV-TO-NEXT(x_2), then SUFF$(x_1, x_2) = 1 + |x_1|$.

Let l_1 and l_2 be two leaves with labels x_1 and x_2. We extend the definition of SUFF to say that SUFF(l_1, l_2) is the same as SUFF(x_1, x_2). We further extend the definition of SUFF to just one node argument. Define SUFF of the rightmost leaf to be ∞. For any other leaf l_1 with right neighbor l_1' define SUFF$(l_1) = (l_1, l_1')$. For an internal node v with leftmost descendant leaf label x^{ℓ} and rightmost descendant leaf label x^{r} define SUFF(v) to be $1 +$ the length of the longest common suffix of PREV-TO-NEXT(v^{ℓ}) and PREV-TO-NEXT(v^{r}), where v^{ℓ} and v^{r} are the leftmost and rightmost descendant leaves of node v, respectively.

Lemma 3.6. *Suppose v is an internal node with children $v_1, v_2, \ldots, v_k$. Then* SUFF$(v) =$ min(min(SUFF(v_1), SUFF(v_1^{r})), min(SUFF(v_2), SUFF(v_2^{r})), …, min(SUFF(v_k), SUFF(v_k^{r}))).

Proof. By construction of the $<_{inv}$ order, the longest common suffix that v^{ℓ} and v^{r} share is also shared by all labels in between v^{ℓ} and v^{r} in the left-to-right leaf order. By the definition of suffix, there must be some pair of consecutive leaves w_i and w_{i+1} whose longest common suffix has the same length (and it is in fact the same common suffix) as that of v^{ℓ} and v^{r}. Thus SUFF(v) = SUFF(v^{ℓ}, v^{r}) = SUFF(w_i) = SUFF(w_i, w_{i+1}). The choice of w_i need not be unique. Suppose w_i descends from the child v_j. If $w_i \neq v_j^{r}$, then w_{i+1} also descends from v_j, SUFF(v) = SUFF(v_j) and all the other contributions to the minimizations are at least as large. Suppose $w_i = v_j^{r}$; then

SUFF(v) = SUFF(v^r_j) and all the other contributions to the minimizations are at least as large. □

For each node v we can either store SUFF(v^r) directly at v or we can store a pointer to v^r and compute SUFF(v^r) with one level of indirection. Given this information, both BALANCE(v) and SUFF(v) can be computed by doing an associative "sum" on the corresponding values at the children of v using $\oplus$ for BALANCE and min for SUFF. Thus when the a–b tree is rebalanced all the nodes that change children can have their BALANCE and SUFF values updated using standard techniques as in [13]. Insertions and deletions in the a–b tree, including function value updates, still take time proportional to the height of the tree, as for plain a–b trees.

In our application, the $<_{inv}$ ordering on the strings automatically ensures that any parenthesis pair we insert or delete preserves the balance in the parenthesis string. The outermost parenthesis pair will correspond to the prefixes (ε, $\#\varepsilon$) on which we never do a *P_Nearest* query; thus every *P_Nearest* query will find an enclosing pair.

One novel feature of our PDDM algorithm is how we insert a node for a new prefix xa into the tree. We search down the a–b tree to figure out where x should go. Our aim is to find the largest prefix smaller than xa in the $<_{inv}$ order on the leaves, so that we know where to insert xa. We start the search at the root of the a–b tree and proceed towards the leaves. Suppose yb is the label associated with an internal node of the tree. At the next level we take the right child of the node if $yb <_{inv} xa$ and the left child otherwise. With this scheme we need $O(\log d)$ tree comparisons to find where to insert xa.

We cannot afford to compare NEXT-TO-PREV(xa) against NEXT-TO-PREV(yb) directly for two reasons. First, we do not explicitly compute or store the NEXT-TO-PREV labels. Even if we did store these labels, they would be arbitrarily long strings and could not be compared in O(1) time. Therefore, we do all the comparisons of x against existing nodes in a roundabout way using the $<_{inv}$ ordering and the rules specified in the following Lemma. We assume that in constant time we can find the last character a of xa, and the parent x of xa in the *pgoto* tree. This can be implemented by making the links in the *pgoto* tree bidirectional.

Lemma 3.7. *To compare xa and yb in the* $<_{inv}$ *order we use the following rules.*

1. *If* $a, b \in \Sigma$ *and* $a < b$, *then* $xa <_{inv} yb$.
2. *If* $a, b \in \Sigma$ *and* $b < a$, *then* $yb <_{inv} xa$.
3. *If* $a = b \in \Sigma$ *then* $xa <_{inv} yb$ *if and only if* $x <_{inv} y$.
4. *If* $a, \in \Sigma$, $b \in \mathbb{N}$, *then* $yb <_{inv} xa$.
5. *If* $a, \in \mathbb{N}$, $b \in \Sigma$, *then* $xa <_{inv} yb$.
6. *If* $a = b \in \mathbb{N}$, *then* $xa <_{inv} yb$ *if and only if* $x <_{inv} y$.
7. *If* $a < b \in \mathbb{N}$, *and* SUFF$(x, y) < a$, *then* $xa <_{inv} yb$ *if and only if* $x <_{inv} y$.
8. *If* $0 \neq a < b \in \mathbb{N}$, *and* SUFF$(x, y) \geqslant a$, *then* $xa <_{inv} yb$.
9. *If* $0 = a < b \in \mathbb{N}$, SUFF$(x, y) \geqslant b$, *and* SUFF$(x, y) < |xa|$, *then* $yb <_{inv} xa$.
10. *If* $0 = a < b \in \mathbb{N}$, SUFF$(x, y) \geqslant b$, *and* SUFF$(x, y) = |xa|$, *then* $xa <_{inv} yb$.

11. *If* $0 = a < b \in \mathbb{N}$, SUFF$(x, y) < b$, *then* $xa <_{inv} yb$ *if and only if* $x <_{inv} y$.
12. *If* $b < a \in \mathbb{N}$, *and* SUFF$(x, y) < b$, *then* $yb <_{inv} xa$ *if and only if* $y <_{inv} x$.
13. *If* $0 \neq b < a \in \mathbb{N}$, *and* SUFF$(x, y) \geqslant b$, *then* $yb <_{inv} xa$.
14. *If* $0 = b < a \in \mathbb{N}$, SUFF$(x, y) \geqslant a$, *and* SUFF$(x, y) > |yb|$, *then* $xa <_{inv} yb$.
15. *If* $0 = b < a \in \mathbb{N}$, SUFF$(x, y) \geqslant a$, *and* SUFF$(x, y) = |yb|$, *then* $yb <_{inv} xa$.
16. *If* $0 = b < a \in \mathbb{N}$, SUFF$(x, y) < a$, *then* $xa <_{inv} xa$ *if and only if* $x <_{inv} y$.

Proof. The first five rules follow from the definition of $<_{inv}$ and our ordering on the characters in Σ and $\mathbb{N}$.

For the rest of the proof assume that $a, b \in \mathbb{N}$. If x is a string in *prev* representation, let $x' =$ PREV-TO-NEXT(x). When PREV-TO-NEXT is applied to xa and yb, the last character will be mapped to a 0 and the character a (or b) places before will be mapped to the value a (or b). All the other characters in $(xa)'$ ($(yb)'$) will be the same as the corresponding characters in x' (y'). The last character of $(xa)'$ and $(yb)'$ is 0 and thus the relative ordering of the two strings cannot be determined from this character. Now we consider Rules 6–11 in five groups.

Rule 6. When $a = b$, the character $a + 1$ places from the right end will be a in both strings. Thus this character is the same in $(xa)'$ and $(yb)'$ and cannot affect the relative ordering of the two strings. Thus the relative ordering of xa and yb is the same as that of x and y.

Rule 7. Suppose that $a < b \in \mathbb{N}$ and SUFF$(x, y) < a$. Note that neither a nor b can be 0. The rightmost position where x' and y' disagree and which determines the relative ordering of x and y comes fewer than a places from their right end (of x', y'). Thus $(xa)'$ and $(yb)'$ will have their rightmost disagreement SUFF$(x, y) + 1$ places from the right end and it will be the same as the rightmost disagreement between x and y. Therefore, xa and yb have the same relative order as x and y in the $<_{inv}$ ordering.

Rule 8. Suppose $0 \neq a < b$ and SUFF$(x, y) \geqslant a$. The placement of an a at the position $a + 1$ places from the right end in $(xa)'$ will cause $(xa)'$ and $(yb)'$ to disagree at that position, which by assumption is at least as far right than where x and y have their rightmost disagreement. Since the a is the largest number in the usual $>$ ordering that $(xa)'$ could have $a + 1$ places from its right end, the other string $(yb)'$ must have either a character in Σ or a smaller number at that position. In either case the ordering we defined on the letters and numbers makes $xa <_{inv} yb$.

Rules 9 and 10. Suppose $0 = a < b \in \mathbb{N}$ and SUFF$(x, y) \geqslant b$. Since $a = 0$, $(xa)' = x'a = x'0$. The last character of $(yb)'$ is also 0. Thus the rightmost place where $(xa)'$ disagrees with $(yb)'$ is $b + 1$ places from the right, where $(yb)'$ has a b. At the place of disagreement, $(xa)'$ must have a bigger character in the $<_{inv}$ order (because it must be a character in Σ or a number that is $< b$ in the usual ordering) unless it has no character there at all, which happens precisely when SUFF$(x, y) = |xa|$. Therefore, we have $yb <_{inv} xa$ if SUFF$(x, y) < |xa|$ (Rule 9) and $xa <_{inv} yb$ if SUFF$(x, y) = |xa|$ (Rule 10).

Rule 11. Suppose $0 = a < b \in \mathbb{N}$, SUFF$(x, y) < b$. In $(yb)'$ the last b is converted to a 0 that matches the last character in $(xa)'$. The b changes the character b spaces from

the right which is beyond (to the left) the rightmost disagreement between x and y. Therefore, $xa <_{inv} yb$ if and only if $x <_{inv} y$.

Rules 12–16 are symmetric to rules 7–11, by reversing the order of a and b. □

Some of the comparison rules require only comparisons between characters and the relative order of other leaves, while others require that we compute SUFF. For the comparisons that do not require SUFF we can gain efficiency by storing the list of leaves in the ordered list data structure of Dietz and Sleator [10]. Using this data structure we can do the comparisons that do not involve SUFF in O(1) time. Our implementation does them in O(1) amortized time. For those comparisons where we need to compute SUFF, we used standard techniques for balanced dynamic trees [9] to compute SUFF and do the comparison in time $O(\log d)$, proportional to the height of the tree.

When we insert a leaf l and its corresponding parenthesis updating BALANCE is straightforward going all the way up the tree using O(1) time per node. We use the following lemma to compute SUFF for l and its neighbor to the left. Just as in insertions of leaves we cannot afford to actually compare entire labels because they are arbitrarily long. Once we compute these two values it is straightforward to update the values of SUFF on the path up the tree in O(1) time per node.

Lemma 3.8. *Suppose that xa and yb are the prev labels of adjacent leaves, then* SUFF(xa, yb) *is given by:*

1. 1 *if* $a, b \in \Sigma$ *and* $a \neq b$.
2. $1 +$ SUFF(x, y), *if* $a = b$.
3. 1 *if* $a \in \Sigma, b \in \mathbb{N}$.
4. 1 *if* $a \in \mathbb{N}, b \in \Sigma$.
5. $1 +$ SUFF(x, y), *if* $a, b \in \mathbb{N}$, $a \neq b$, *and* SUFF$(x, y) < \min(a, b)$.
6. $1 + \min(a, b)$, *if* $a, b \in \mathbb{N}$, $a \neq b$, *and* SUFF$(x, y) \geqslant \min(a, b) > 0$.
7. $1 + \min(\max(a, b),$ SUFF$(x, y))$ *if* $a, b \in \mathbb{N}$, $a \neq b$, *and* $\min(a, b) = 0$.

Proof. When a or b is in Σ the proof is straightforward because *prev*, *next*, and PREV-TO-NEXT are the identity on Σ. The following cases remain.

Rule 2. If $a = b \in \mathbb{N}$, then in both xa and yb, PREV-TO-NEXT maps the last character to 0, and maps the number $a + 1$ places from the right end to an a. Thus the two characters affected by the addition of the a match in both xa and yb and the character $a + 1$ spaces from the right end cannot cause the longest common subsequence to shorten. In both x' and y', the characters a spaces from the right must be a 0 and thus they match there too. Therefore by adding a to x and to y, the value of SUFF increases by exactly 1.

Rule 5. If $a, b \in \mathbb{N}$ and SUFF$(x, y) < \min(a, b)$, then the change $a + 1$ or $b + 1$ places away from the right end, does not occur in the short suffix where x' and y' agree. The common suffix is extended by 1 because PREV-TO-NEXT maps both the final a in xa and the final b in yb to 0.

Rule 6. If $a, b \in \mathbb{N}$, $a \neq b$, and $\text{SUFF}(x, y) \geqslant \min(a, b) > 0$, then the change away from the right end does occur in the longest suffix shared by x' and y'. Thus all the rightmost $min(a, b)$ *characters in* $(xa)'$ and $(yb)'$ must still agree. The final characters in the two strings agree because they are both 0. Thus $\text{SUFF}(xa, yb) = 1 + \min(a, b)$.

Rule 7. Suppose $a, b \in \mathbb{N}$, $a \neq b$ and $\min(a, b) = 0$. Suppose $a = 0$; the other case is symmetric. By this assumption $(xa)' = x'a$. In yb, the addition of b changes the character $b + 1$ places from the right in $(yb)'$ into a b which certainly cannot match the corresponding character in $(xa)'$. Therefore, the longest common suffix ends either $b + 1$ characters from the right or wherever it ended for x and y, whichever is sooner. That is $\text{SUFF}(xa, yb) = 1 + \min(b, \text{SUFF}(x, y))$. When we do not know which of a or b is 0, we get that $\text{SUFF}(xa, yb) = 1 + \min(\max(a, b), \text{SUFF}(x, y))$. □

We now give the pseudocode for SEARCH, INSERT, and DELETE, which are used for searching a text, inserting the prefixes of a pattern in the a–b tree, and deleting the prefixes of a pattern from the a–b tree, respectively. We use prefixes instead of states for clarity.

Algorithm 1. Pseudocode for searching a text.

```
SEARCH(T = t_1 ... t_n)
  state ← ε
  for i ← 1 to n do
    while pgoto(state, t_i) is undefined
      state ← pfail(state)
    state ← pgoto(state, t_i)
    /* Pretend a $ is read to check if any pattern match */
    temp ← pgoto(state, $) if defined, otherwise temp ← state·$
    If temp is not normal then temp ← pfail(temp)
    while temp ≠ $ do /* Report all nonempty patterns */
      /* Since temp ends in $ we have matched a pattern */
      Print the pattern that temp point to
      temp ← pfail(temp)/* See if any smaller patterns match */
```

Algorithm 2. Pseudocode for inserting a pattern into the tree for a dictionary.

```
INSERT(P = p_1...p_m)/* last character p_m is always $ */
  Suppose p_1...p_j is the longest prefix of P shared by some other pattern.
  Increment the reference count for the prefixes of p_1...p_j.
  For i ← j + 1 to m do
  Let x = p_1...p_{i-1}. Let a = p_i. /* xa is being inserted. x is already in S */
  Insert a leaf and left parenthesis for xa updating BALANCE and SUFF values as
                                                                    needed
```

Insert a leaf and right parenthesis $\#xa$ updating BALANCE and SUFF values as needed

Insert a leaf for $xa\$$ and a left parenthesis if a is the last character updating BALANCE and SUFF values as needed

Insert a leaf for $\#xa\$$ and a right parenthesis if a is the last character updating BALANCE and SUFF values as needed

Algorithm 3. Pseudocode for deleting a pattern from the tree for a dictionary.

DELETE($P = p_1 \ldots p_m$) /* $p_m = \$$ */

Suppose $p_1 \ldots p_j$ is the longest prefix of P shared by some other pattern.

Decrement the reference count for the prefixes of $p \ldots p_j$.

For $i \leftarrow m$ downto $j + 1$ do

Let $x = p_1 \ldots p_i$ /* x is a normal prefix */

If $x\$$ is still in S Then

delete $x\$$ and $\#x\$$ updating BALANCE and SUFF values as needed

delete x and $\#x$ updating BALANCE and SUFF values as needed

Theorem 3.9. *Let D be a dictionary of parameterized patterns over an alphabet $\Sigma \cup \Pi$. We can search a parameterized text T for patterns of D in time $O((t + tocc) \cdot (\log \sigma + \log d))$, where tocc is the total number of patterns reported. We can insert a pattern P in time $O(p \cdot (\log \sigma + \log^2 d))$. We can delete a pattern P in time $O(p \cdot (\log \sigma + \log d))$. Moreover, we require only $O(d)$ space to store the automaton.*

Proof. The search algorithm is essentially the same as in the nondynamic case, but we have a different representation of *pfail*. As long as we can maintain the $<_{inv}$ order on the prefixes, the equivalence of *pfail* computation and nearest enclosing parentheses queries holds. The insertion and deletion algorithms use a standard approach for inserting and deleting items from a list stored as leaves of a balanced tree. The main differences from the standard algorithms are how to do the comparisons for insert and how to update the SUFF and BALANCE information. The algorithm for comparisons was proved correct in Lemma 3.7, the update of the SUFF information was proved correct in Lemma 3.8, and the update of the BALANCE information was proved correct in [13].

The running time analysis is similar to the static case. The extra log factor in searching comes from the cost of computing *pfail*. One extra log factor in the insertion and deletion cost comes from the cost of updating the tree which is proportional to the height of the tree. The second extra log factor in the insertion cost comes from the cost of doing a comparison. □

In our implementation, we preprocessed the initial dictionary by insertion of the individual patterns.

References

[1] A.V. Aho and M.J. Corasick, Efficient string matching: an aid to bibliographic search, *Commun. ACM* **18** (1975) 333–340.

[2] B. Alpern, R. Hoover, B.K. Rosen, P.F. Sweeney and F.K. Zadeck, Incremental evaluation of computational circuits, *Proc 1st Ann. ACM–SIAM Symp. on Discrete Algorithms* (1990) 32–42.

[3] A. Amir and M. Farach, Adaptive dictionary matching, *Proc. 32nd IEEE Ann Symp. on Foundation of Computer Science* (1991) 760–766.

[4] A. Amir, M. Farach, R. Giancarlo, Z. Galil and K. Park, Dynamic dictionary matching, *J. Comput. Systems Sci.* **49** (1994) 208–222.

[5] A. Amir, M. Farach, R.M. Idury, J.A. La Poutré and A.A. Schäffer, Improved dynamic dictionary matching, *Proc. 4th Ann. ACM–SIAM Symp. on Discrete Algorithms* (1993) 392–401. Full paper to appear in *Inform. Comput.*

[6] A. Amir, M. Farach and S. Muthukrishnan, Alphabet dependence in parameterized matching, *Inform. Process. Lett.* **49** (1994) 111–115.

[7] B. Baker, A theory of parameterized pattern matching: algorithms and applications, *Proc. 25th Ann. ACM Symp. on Theory of Computing* (1993) 71–80.

[8] M.T. Chen and J. Seiferas, Efficient and elegant subword tree construction, in: A. Apostolico and Z. Galil, eds., *Combinatorial Algorithms on Words*, ch. 12, NATO ASI Series F: Computer and System Sciences (1985) 97–107.

[9] T.H. Cormen, C.E. Leiserson and R.L. Rivest, *Introduction to Algorithms* (MIT Press, Cambridge, 1990).

[10] P. Dietz and D.D Sleator, Two algorithms for maintaining order in a list, in: *Proc. 19th Ann. ACM Symp. on Theory of Computing*, (1987) 365–372.

[11] J.R. Driscoll, N. Sarnak, D.D. Sleator and R.E. Tarjan, Making data structures persistent, *J. Comput Systems Sci.* **38** (1989) 86–124.

[12] Z. Galil and G.F. Italiano, A note on set union with arbitrary deunions, *Inform. Proc. Lett.* **37** (1991) 331–335.

[13] R.H. Güting and D. Wood, The parenthesis tree, *Inform. Sci.* **27** (1982) 151–162.

[14] R.M. Idury, Dynamic multiple pattern matching, Ph.D. thesis, Rice University, 1993.

[15] R.M. Idury and A.A. Schäffer, Dynamic dictionary matching with failure functions. in: Proc. 3rd *Symp. on Combinatorial Pattern Matching*, Lecture Notes in Computer Science Vol. 644 (Springer, Berlin, 1992) 276–287. Full paper *Theoret. Comput. Sci.* **131** (1994) 295–310.

[16] D.E. Knuth, J.H. Morris and V.R. Pratt, Fast pattern matching in strings, *SIAM J. Comput.* **6** (1977) 323–350.

[17] E.M. McCreight, A space-economical suffix tree construction algorithm, *J. ACM* **23** (1976) 262–272.

[18] K. Mehlhorn, *Data Structures and Algorithms* 1: *Sorting and Searching* (Springer, Berlin 1984).

[19] B. Meyer, Incremental string matching, *Inform. Process. Lett.* **21** (1985) 219–227.

[20] P. Weiner, Linear pattern matching algorithm, *Proc. 14th IEEE Ann. Symp. on Switching and Automata Theory* (1973) 1–11.